Date=12/10/2020
Lecture By=Manish Mahant
Subject ⇒Revision Classes

Overview For The Lecture

What is ES5?

ES5 is a shortcut for ECMAScript 5

ECMAScript 5 is also known as JavaScript 5

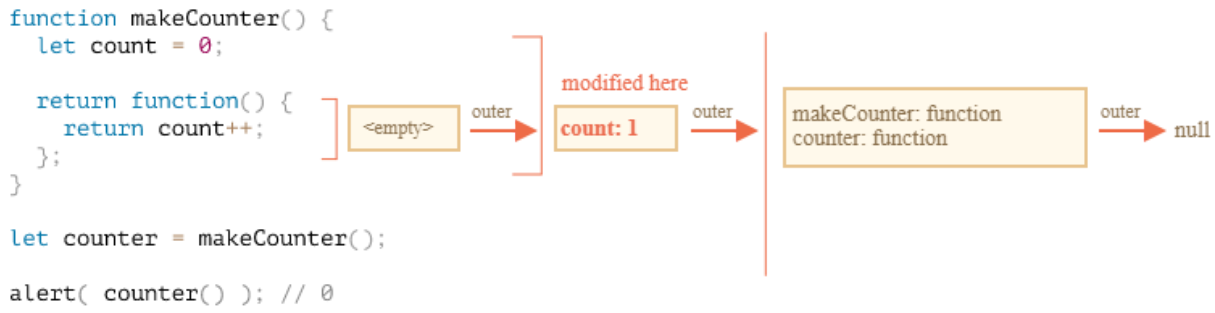
ECMAScript 5 is also known as ECMAScript 2009

ECMAScript 5 Syntactical Changes

- Property access [ ] on strings
- Trailing commas in array and object literals
- Multiline string literals
- Reserved words as property names

Closure In Javascript

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.



Lexical scoping

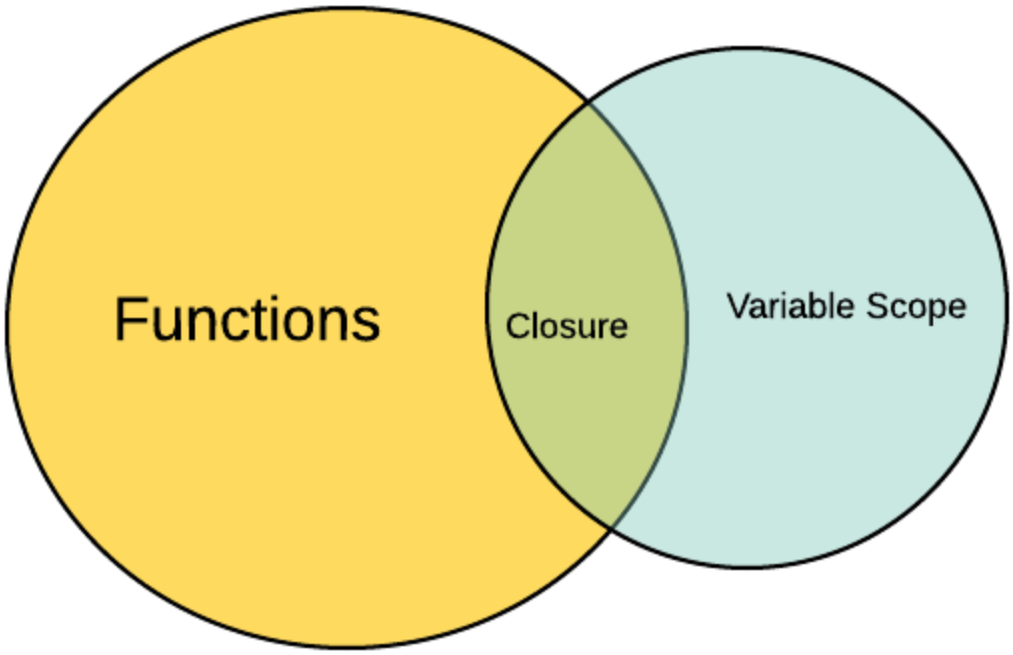
Consider the following example code:

```
function init() {
  var name = 'Mozilla'; // name is a local variable created by init
  function displayName() { // displayName() is the inner function, a
closure
    alert(name); // use variable declared in the parent function
  }
  displayName();
}
init();
```

`init()` creates a local variable called `name` and a function called `displayName()`. The `displayName()` function is an inner function that is defined inside `init()` and is

available only within the body of the `init()` function. Note that the `displayName()` function has no local variables of its own. However, since inner functions have access to the variables of outer functions, `displayName()` can access the variable `name` declared in the parent function, `init()`.

Closure



Consider the following code example:

```
function makeFunc() {
  var name = 'Mozilla';
  function displayName() {
    alert(name);
  }
  return displayName;
}
```

```
var myFunc = makeFunc();
myFunc();
```

Running this code has exactly the same effect as the previous example of the `init()` function above. What's different (and interesting) is that the `displayName()` inner function is returned from the outer function *before being executed*.

At first glance, it might seem unintuitive that this code still works. In some programming languages, the local variables within a function exist for just the duration of that function's execution. Once `makeFunc()` finishes executing, you might expect that the name variable would no longer be accessible. However, because the code still works as expected, this is obviously not the case in JavaScript.

Here's a slightly more interesting example—a `makeAdder` function:

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

Emulating private methods with closures

Languages such as Java allow you to declare methods as private, meaning that they can be called only by other methods in the same class.

JavaScript does not provide a native way of doing this, but it is possible to emulate private methods using closures. Private methods aren't just useful for restricting access to code. They also provide a powerful way of managing your global namespace.

The following code illustrates how to use closures to define public functions that can access private functions and variables. Note that these closures follow the Module Design Pattern.

```
var counter = (function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }

  return {
    increment: function() {
      changeBy(1);
    },

    decrement: function() {
      changeBy(-1);
    },

    value: function() {
      return privateCounter;
    }
  };
})();

console.log(counter.value());  // 0.

counter.increment();
counter.increment();
console.log(counter.value());  // 2.

counter.decrement();
console.log(counter.value());  // 1.
```

Notice that I defined an anonymous function that creates a counter, and then I call it immediately and assign the result to the `counter` variable. You could store this function in a separate variable `makeCounter`, and then use it to create several counters

```
var makeCounter = function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },

    decrement: function() {
      changeBy(-1);
    },

    value: function() {
      return privateCounter;
    }
  }
};

var counter1 = makeCounter();
var counter2 = makeCounter();

alert(counter1.value());  // 0.

counter1.increment();
counter1.increment();
alert(counter1.value()); // 2.

counter1.decrement();
alert(counter1.value()); // 1.
alert(counter2.value()); // 0.
```

Closure Scope Chain

Every closure has three scopes:

- Local Scope (Own scope)
- Outer Functions Scope
- Global Scope

A common mistake is not realizing that, in the case where the outer function is itself a nested function, access to the outer function's scope includes the enclosing scope of the outer function—effectively creating a chain of function scopes. To demonstrate, consider the following example code.

Questions For Self Practice

https://au-assignment.s3.ap-south-1.amazonaws.com/Week_16_Day_2_Challenge-260af8bd-f521-4128-b337-9320f1d75522.pdf

Resource For The Class

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

Youtube Tutorial

https://www.youtube.com/watch?v=71AtaJpJHw0